Capítulo 4

ALGORITMOS ÁVIDOS

4.1 INTRODUCCIÓN

El método que produce algoritmos ávidos es un método muy sencillo y que puede ser aplicado a numerosos problemas, especialmente los de optimización.

Dado un problema con *n* entradas el método consiste en obtener un subconjunto de éstas que satisfaga una determinada restricción definida para el problema. Cada uno de los subconjuntos que cumplan las restricciones diremos que son soluciones *prometedoras*. Una solución prometedora que maximice o minimice una función objetivo la denominaremos solución óptima.

Como ayuda para identificar si un problema es susceptible de ser resuelto por un algoritmo ávido vamos a definir una serie de elementos que han de estar presentes en el problema:

- Un conjunto de *candidatos*, que corresponden a las *n* entradas del problema.
- Una *función de selección* que en cada momento determine el candidato idóneo para formar la solución de entre los que aún no han sido seleccionados ni rechazados.
- Una función que compruebe si un cierto subconjunto de candidatos es *prometedor*. Entendemos por prometedor que sea posible seguir añadiendo candidatos y encontrar una solución.
- Una *función objetivo* que determine el valor de la solución hallada. Es la función que queremos maximizar o minimizar.
- Una función que compruebe si un subconjunto de estas entradas es solución al problema, sea óptima o no.

Con estos elementos, podemos resumir el funcionamiento de los algoritmos ávidos en los siguientes puntos:

1. Para resolver el problema, un algoritmo ávido tratará de encontrar un subconjunto de candidatos tales que, cumpliendo las restricciones del problema, constituya la solución óptima.
2. Para ello trabajará por etapas, tomando en cada una de ellas la decisión que le parece la mejor, sin considerar las consecuencias futuras, y por tanto escogerá

de entre todos los candidatos el que produce un óptimo local para esa etapa, suponiendo que será a su vez óptimo global para el problema.

3. Antes de añadir un candidato a la solución que está construyendo comprobará si es prometedora al añadilo. En caso afirmativo lo incluirá en ella y en caso contrario descartará este candidato para siempre y no volverá a considerarlo.
4. Cada vez que se incluye un candidato comprobará si el conjunto obtenido es solución.

Resumiendo, los algoritmos ávidos construyen la solución en etapas sucesivas, tratando siempre de tomar la decisión óptima para cada etapa. A la vista de todo esto no resulta difícil plantear un esquema general para este tipo de algoritmos:

```
PROCEDURE AlgoritmoAvido(entrada:CONJUNTO):CONJUNTO;
   VAR x:ELEMENTO; solucion:CONJUNTO; encontrada:BOOLEAN;
BEGIN
   encontrada:=FALSE; crear(solucion);
   WHILE NOT EsVacio(entrada) AND (NOT encontrada) DO
      x:=SeleccionarCandidato(entrada);
      IF EsPrometedor(x,solucion) THEN
         Incluir(x,solucion);
         IF EsSolucion(solucion) THEN
            encontrada:=TRUE
         END;
      END
   END;
   RETURN solucion;
END AlgoritmoAvido;
```

De este esquema se desprende que los algoritmos ávidos son muy fáciles de implementar y producen soluciones muy eficientes. Entonces cabe preguntarse ¿por qué no utilizarlos siempre? En primer lugar, porque no todos los problemas admiten esta estrategia de solución. De hecho, la búsqueda de óptimos locales no tiene por qué conducir siempre a un óptimo global, como mostraremos en varios ejemplos de este capítulo. La estrategia de los algoritmos ávidos consiste en tratar de ganar todas las batallas sin pensar que, como bien saben los estrategas militares y los jugadores de ajedrez, para ganar la guerra muchas veces es necesario perder alguna batalla.

Desgraciadamente, y como en la vida misma, pocos hechos hay para los que podamos afirmar sin miedo a equivocarnos que lo que parece bueno para hoy siempre es bueno para el futuro. Y aquí radica la dificultad de estos algoritmos. Encontrar la función de selección que nos garantice que el candidato escogido o rechazado en un momento determinado es el que ha de formar parte o no de la solución óptima sin posibilidad de reconsiderar dicha decisión. Por ello, una parte muy importante de este tipo de algoritmos es la demostración formal de que la función de selección escogida consigue encontrar óptimos globales para cualquier entrada del algoritmo. No basta con diseñar un procedimiento ávido, que seguro

que será rápido y eficiente (en tiempo y en recursos), sino que hay que demostrar que siempre consigue encontrar la solución óptima del problema.

Debido a su eficiencia, este tipo de algoritmos es muchas veces utilizado aun en los casos donde se sabe que no necesariamente encuentran la solución óptima. En algunas ocasiones la situación nos obliga a encontrar pronto una solución razonablemente buena, aunque no sea la óptima, puesto que si la solución óptima se consigue demasiado tarde, ya no vale para nada (piénsese en el localizador de un avión de combate, o en los procesos de toma de decisiones de una central nuclear). También hay otras circunstancias, como veremos en el capítulo dedicado a los algoritmos que siguen la técnica de Ramificación y Poda, en donde lo que interesa es conseguir cuanto antes una solución del problema y, a partir de la información suministrada por ella, conseguir la óptima más rápidamente. Es decir, la eficiencia de este tipo de algoritmos hace que se utilicen aunque no consigan resolver el problema de optimización planteado, sino que sólo den una solución "aproximada".

El nombre de algoritmos ávidos, también conocidos como voraces (su nombre original proviene del término inglés *greedy*) se debe a su comportamiento: en cada etapa "toman lo que pueden" sin analizar consecuencias, es decir, son glotones por naturaleza. En lo que sigue veremos un conjunto de problemas que muestran cómo diseñar algoritmos ávidos y cuál es su comportamiento. En este tipo de algoritmos el proceso no acaba cuando disponemos de la implementación del procedimiento que lo lleva a cabo. Lo importante es la demostración de que el algoritmo encuentra la solución óptima en todos los casos, o bien la presentación de un contraejemplo que muestra los casos en donde falla.

4.2 EL PROBLEMA DEL CAMBIO

Suponiendo que el sistema monetario de un país está formado por monedas de valores $v_1,v_2,...,v_n$, el problema del cambio de dinero consiste en descomponer cualquier cantidad dada M en monedas de ese país utilizando el menor número posible de monedas.

En primer lugar, es fácil implementar un algoritmo ávido para resolver este problema, que es el que sigue el proceso que usualmente utilizamos en nuestra vida diaria. Sin embargo, tal algoritmo va a depender del sistema monetario utilizado y por ello vamos a plantearnos dos situaciones para las cuales deseamos conocer si el algoritmo ávido encuentra siempre la solución óptima:

a) Suponiendo que cada moneda del sistema monetario del país vale al menos el doble que la moneda de valor inferior, que existe una moneda de valor unitario, y que disponemos de un número ilimitado de monedas de cada valor.

b) Suponiendo que el sistema monetario está compuesto por monedas de valores 1, $p, p^2, p^3,..., p^n$, donde $p > 1$ y $n > 0$, y que también disponemos de un número ilimitado de monedas de cada valor.

Solución (👓)

Comenzaremos con la implementación de un algoritmo ávido que resuelve el problema del cambio de dinero:

```
TYPE  MONEDAS =(M500,M200,M100,M50,M25,M5,M1);(*sistema monetario*)
      VALORES = ARRAY MONEDAS OF CARDINAL; (* valores de monedas *)
      SOLUCION = ARRAY MONEDAS OF CARDINAL;

PROCEDURE Cambio(n:CARDINAL;VAR valor:VALORES;VAR cambio:SOLUCION);
(* n es la cantidad a descomponer, y el vector "valor" contiene los
valores de cada una de las monedas del sistema monetario *)
   VAR moneda:MONEDAS;
BEGIN
   FOR moneda:=FIRST(MONEDAS) TO LAST(MONEDAS) DO
      cambio[moneda]:=0
   END;
   FOR moneda:=FIRST(MONEDAS) TO LAST(MONEDAS) DO
      WHILE valor[moneda]<=n DO
         INC(cambio[moneda]);
         DEC(n,valor[moneda])
      END
   END
END Cambio;
```

Este algoritmo es de complejidad lineal respecto al número de monedas del país, y por tanto muy eficiente.

Respecto a las dos cuestiones planteadas, comenzaremos por la primera. Supongamos que nuestro sistema monetario esta compuesto por las siguientes monedas:

```
TYPE MONEDAS = (M11,M5,M1);  valor:={11,5,1};
```

Tal sistema verifica las condiciones del enunciado pues disponemos de moneda de valor unitario, y cada una de ellas vale más del doble de la moneda inmediatamente inferior.

Consideremos la cantidad $n = 15$. El algoritmo ávido del cambio de monedas descompone tal cantidad en:

$$15 = 11 + 1 + 1 + 1 + 1,$$

es decir, mediante el uso de cinco monedas. Sin embargo, existe una descomposición que utiliza menos monedas (exactamente tres):

$$15 = 5 + 5 + 5.$$

Aunque queda comprobado que bajo estas circunstancias el diseño ávido no puede utilizarse, las razones por las que el algoritmo falla quedarán al descubierto cuando analicemos el siguiente punto.

b) En cuanto a la segunda situación, y para demostrar que el algoritmo ávido encuentra la solución óptima, vamos a apoyarnos en una propiedad general de los números naturales:

Si p es un número natural mayor que 1, todo número natural x puede expresarse de forma única como:

$$x = r_0 + r_1p + r_2p^2 + ... + r_np^n, \qquad [4.1]$$

con $0 \le r_i < p$ para todo $0 \le i \le n$ y siendo n el menor natural tal que $x < p^{n+1}$, es decir, $n = \lfloor \log_p x \rfloor$.

El algoritmo del cambio de monedas lo que hace en nuestro caso es calcular los r_i, que indican el número de monedas a devolver de valor p^i ($0 \le i \le n$). Lo que tenemos que demostrar es que esa descomposición es óptima, esto es, que si

$$x = s_0 + s_1p + s_2p^2 + ... + s_mp^m$$

es otra descomposición distinta, entonces:

$$\sum_{i=0}^{n} r_i < \sum_{i=0}^{m} s_i .$$

Para realizar esta demostración lo haremos primero para $p = 2$ porque intuitivamente resulta más sencillo de entender el proceso de la demostración. El caso general resulta ser análogo.

Sea entonces

$$x = r_0 + 2r_1 + 2^2r_2 + ... + 2^nr_n \qquad [4.2]$$

la descomposición obtenida por el algoritmo ávido. Por tanto $x < 2^{n+1}$ y los coeficientes r_i toman los valores 0 ó 1. Consideremos además otra descomposición distinta:

$$x = s_0 + 2s_1 + 2^2s_2 + ... + 2^m s_m.$$

Paso 1:

En primer lugar, como se verifica que $x < 2^{n+1}$, esto implica que $m \le n$. Definimos entonces $s_{m+1} = s_{m+2} = ... = s_n = 0$ para poder disponer de n términos en cada descomposición.

Paso 2:

Queremos ver que

$$r_0 + r_1 + ... + r_n < s_0 + s_1 + ... + s_n.$$

Como ambas descomposiciones son distintas, sea k el primer índice tal que $r_k \ne s_k$. Podemos suponer sin perder generalidad que $k = 0$, puesto que si no lo fuera podríamos restar a ambos lados de la desigualdad los términos iguales y dividir por la potencia de 2 adecuada. Veamos que si $r_0 \ne s_0$ entonces $r_0 < s_0$.

- Si x es par entonces $r_0 = 0$. Como $s_0 \ge 0$ y estamos suponiendo que $r_0 \ne s_0$, s_0 ha de ser mayor que cero y por tanto podemos deducir que $r_0 < s_0$.

- Si x es impar entonces $r_0 = 1$. Pero en la segunda descomposición de x también ha de haber al menos una moneda de una unidad, y por tanto $s_0 \geq 1$. Al estar suponiendo que $r_0 \neq s_0$, podemos deducir también aquí que $r_0 < s_0$.

Con esto, consideremos la cantidad $s_0 - r_0 > 0$. Tal cantidad ha de ser par pues $x - r_0$ lo es (por la expresión [4.2]). Y por ser par, siempre podremos "mejorar" la segunda descomposición $(s_0,s_1,...,s_n)$ cambiando $s_0 - r_0$ monedas de 1 unidad por $(s_0 - r_0)/2$ monedas de 2 unidades, obteniendo:

$$s_0 + s_1 + \ldots + s_n > r_0 + \left(s_1 + \frac{s_0 - r_0}{2} \right) + s_2 + \ldots + s_n . \qquad [4.3]$$

Paso 3:

Mediante el razonamiento anterior hemos obtenido una nueva descomposición, mejor que la segunda, y manteniendo además que:

$$r_0 + \left(s_1 + \frac{s_0 - r_0}{2} \right) 2 + s_2 2^2 + \ldots + s_n 2^n = x = r_0 + r_1 2 + \ldots + r_n 2^n .$$

Podemos volver a aplicar el razonamiento del paso 2 sobre esta nueva descomposición, y así sucesivamente ir viendo que $s_i \geq r_i$ para todo $0 \leq i \leq n-1$, e ir obteniendo nuevas descomposiciones, cada una mejor que la anterior, hasta llegar en el último paso a una descomposición de la forma:

$$r_0 + r_1 2 + r_2 2^2 + \ldots + r_{n-1} 2^{n-1} + \left(s_n + \frac{s_{n-1} - acum_{n-1}}{2} \right) 2^n = x \qquad [4.4]$$

en la que hemos ido acumulando las diferencias en el último término, y que además verifica que:

$$r_0 + r_1 + \ldots + r_i + \left(s_{i+1} + \frac{s_i - acum_i}{2} \right) + \ldots + s_n \geq r_0 + r_1 + \ldots + r_n , \ (0 \leq i \leq n-1)$$

Una vez llegado a este punto la demostración está ya realizada, puesto que si se verifica [4.4], por la unicidad de la descomposición a la que hacía referencia la propiedad [4.1], se ha de cumplir que

$$\left(s_n + \frac{s_{n-1} - acum_{n-1}}{2} \right) = r_n ,$$

y esto, junto a la cadena de desigualdades [4.3], hace que sea cierta nuestra afirmación. Para el caso $p > 2$ el razonamiento es igual.

Resta sólo preguntarnos por qué esta demostración no funciona para cualquier sistema monetario. La razón fundamental se encuentra en la expresión [4.4], que en este sistema permite pasar monedas de una unidad a otra sin problemas, no siendo válido para todos.

4.3 RECORRIDOS DEL CABALLO DE AJEDREZ

Dado un tablero de ajedrez y una casilla inicial, queremos decidir si es posible que un caballo recorra todos y cada uno de los escaques sin duplicar ninguno. No es necesario en este problema que el caballo vuelva al escaque de partida. Un posible algoritmo ávido decide, en cada iteración, colocar el caballo en la casilla desde la cual domina el menor número posible de casillas aún no visitadas.

a) Implementar dicho algoritmo a partir de un tamaño de tablero $n \times n$ y una casilla inicial (x_0,y_0).

b) Buscar, utilizando el algoritmo realizado en el apartado anterior, todas las casillas iniciales para los que el algoritmo encuentra solución.

c) Basándose en los resultados del apartado anterior, encontrar el patrón general de las soluciones del recorrido del caballo.

Solución (☺/😐)

a) Para implementar el algoritmo pedido comenzaremos definiendo las constantes y tipos que utilizaremos:

```
CONST TAMMAX = ...; (* dimension maxima del tablero *)
TYPE tablero = ARRAY[1..TAMMAX],[1..TAMMAX] OF CARDINAL;
```

Cada una de las casillas del tablero va a almacenar un número natural que indica el número de orden del movimiento del caballo en el que visita la casilla. Podrá tomar también el valor cero, indicando que la casilla no ha sido visitada aún. Inicialmente todas las casillas tomarán este valor.

Una posible implementación del algoritmo viene dada por la función *Caballo* que se muestra a continuación, la cual, dado un tablero *t*, su dimensión *n* y una posición inicial (x,y), decide si el caballo recorre todo el tablero o no.

```
PROCEDURE Caballo(VAR t:tablero; n:CARDINAL; x,y:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   InicTablero(t,n); (* inicializa las casillas del tablero a 0 *)
   FOR i:=1 TO n*n DO
      t[x,y]:=i;
      IF NOT NuevoMov(t,n,x,y) AND (i<n*n-1) THEN RETURN FALSE END;
   END;
   RETURN TRUE; (* hemos recorrido las n*n casillas *)
END Caballo;
```

La función *NuevoMov* es la que va a ir calculando la nueva casilla a la que salta el caballo siguiendo la indicación del enunciado, devolviendo *FALSE* si no puede moverse:

```
PROCEDURE NuevoMov(VAR t:tablero; n:CARDINAL; VAR x,y:CARDINAL)
                                                       :BOOLEAN;
```

```
  VAR accesibles,minaccesibles:CARDINAL;
      i,solx,soly,nuevax,nuevay:CARDINAL;
BEGIN
  minaccesibles:=9;
  solx:=x; soly:=y;
  FOR i:=1 TO 8 DO
    IF Salto(t,n,i,x,y,nuevax,nuevay) THEN
      accesibles:=Cuenta(t,n,nuevax,nuevay);
      IF (accesibles>0) AND (accesibles<minaccesibles) THEN
        minaccesibles:=accesibles;
        solx:=nuevax; soly:=nuevay;
      END
    END
  END;
  x:=solx; y:=soly;
  RETURN (minaccesibles<9);
END NuevoMov;
```

Para su implementación necesitamos dos funciones auxiliares: *Salto* y *Cuenta*. La primera calcula las coordenadas de la casilla a donde salta el caballo (tiene 8 posibilidades), y devuelve si es posible realizar ese movimiento o no (puede estar ocupada o bien salirse del tablero):

```
PROCEDURE Salto(VAR t:tablero;n:CARDINAL;i:CARDINAL;
                x,y:CARDINAL;VAR nx,ny:CARDINAL) :BOOLEAN;
  (* i indica el numero del movimiento, (x,y) es la casilla
     actual, y (nx,ny) es la casilla a donde salta. *)
BEGIN
  CASE i OF
    |1: nx:=x-2; ny:=y+1;  |2: nx:=x-1; ny:=y+2;
    |3: nx:=x+1; ny:=y+2;  |4: nx:=x+2; ny:=y+1;
    |5: nx:=x+2; ny:=y-1;  |6: nx:=x+1; ny:=y-2;
    |7: nx:=x-1; ny:=y-2;  |8: nx:=x-2; ny:=y-1;
  END;
  RETURN((1<=nx)AND(nx<=n)AND(1<=ny)AND(ny<=n)AND(t[nx,ny]=0));
END Salto;
```

Dicha función intenta los movimientos en el orden que muestra la siguiente figura:

	2		3	
1				4
		X		

8				5
	7		6	

La otra función es *Cuenta*, que devuelve el número de casillas a las que el caballo puede saltar desde una posición dada:

```
PROCEDURE Cuenta(VAR t:tablero;n:CARDINAL;x,y:CARDINAL):CARDINAL;
   VAR acc,i,nx,ny:CARDINAL;
BEGIN
   acc:=0;
   FOR i:=1 TO 8 DO
      IF Salto(t,n,i,x,y,nx,ny) THEN INC(acc) END
   END;
   RETURN acc;
END Cuenta;
```

Obsérvese que hemos utilizado el paso del tablero por referencia (mediante *VAR*) en vez de por valor en todos los procedimientos aunque no se modifique el vector, para evitar su copia en la pila.

b) Para resolver esta cuestión necesitamos un programa que nos permita ir recorriendo todas las posibilidades e imprimiendo aquellas casillas iniciales desde donde se consigue solución:

```
MODULE Caballos;
   ....
   VAR t:tablero; n,i,j:CARDINAL;
BEGIN
   FOR n:=4 TO TAMMAX DO
      WrStr('Dimension = '); WrCard(n,0); WrLn();
      FOR i:=1 TO n DO FOR j:=1 TO n DO
            IF Caballo(t,n,i,j) THEN
               WrStr(' Desde: '); WrCard(i,0); WrStr(',');
               WrCard(j,0); WrStr(' tiene solucion.'); WrLn();
            END
      END END;
      WrLn();
   END
END Caballos.
```

c) La salida del programa anterior nos permite inferir un patrón general para las soluciones del problema:

- Para $n = 4$, el problema no tiene solución.
- Para $n > 4$, n par, el problema tiene solución para cualquier casilla inicial.

- Para $n > 4$, n impar, el problema tiene solución para aquellas casillas iniciales (x_0,y_0) que verifiquen que $x_0 + y_0$ sea par, es decir, si el caballo comienza su recorrido en una escaque blanco.

Pero observemos que el algoritmo implementado no ha encontrado solución en todas estas situaciones. Por ejemplo, para $n = 5$, $x_0 = 5$ e $y_0 = 3$ el programa dice que no la hay. Sin embargo, sí la encuentra para $n = 5$, $x_0 = 1$ e $y_0 = 3$, para $n = 5$, $x_0 = 3$ e $y_0 = 1$ y para $n = 5$, $x_0 = 3$ e $y_0 = 5$, que son casos simétricos. De existir solución para alguno de ellos, por simetría se obtiene para los otros. ¿Por qué no la encuentra nuestro algoritmo?

La respuesta a esta pregunta se encuentra en cómo buscamos la siguiente casilla a donde saltar. Por la forma en la que funciona el programa, *siempre* probamos las ocho casillas en el sentido de las agujas del reloj, siguiendo la pauta mostrada en la función *Salto*. Esto hace que nuestro algoritmo no sea simétrico. En resumen, estamos ante un algoritmo ávido que no funciona para todos los casos.

4.4 LA DIVISIÓN EN PÁRRAFOS

Dada una secuencia de palabras $p_1, p_2, ..., p_n$ de longitudes $l_1, l_2, ..., l_n$ se desea agruparlas en líneas de longitud L. Las palabras están separadas por espacios cuya amplitud ideal (en milímetros) es b, pero los espacios pueden reducirse o ampliarse si es necesario (aunque sin solapamiento de palabras), de tal forma que una línea $p_i, p_{i+1}, ..., p_j$ tenga exactamente longitud L. Sin embargo, existe una penalización por reducción o ampliación en el número total de espacios que aparecen o desaparecen. El *costo* de fijar la línea $p_i, p_{i+1}, ..., p_j$ es $(j - i)|b' - b|$, siendo b' el ancho real de los espacios, es decir $(L - l_i - l_{i+1} - ... - l_j)/(j - i)$. No obstante, si $j = n$ (la última palabra) el costo será cero a menos que $b' < b$ (ya que no es necesario ampliar la última línea).

En primer lugar, necesitamos plantear un algoritmo ávido para resolver el problema, implementarlo y dar un ejemplo donde este algoritmo no encuentre solución óptima o bien demostrar que tal ejemplo no existe.

Por otra lado, consideraremos el caso especial de usar una impresora de líneas, en donde por sus características especiales el valor óptimo de b es 1 y no se puede producir reducción de espacios (ya que b no puede ser 0).

Solución (☺)

Para resolver este problema mediante un algoritmo ávido pensemos en lo que haríamos en la práctica para solucionarlo. En primer lugar, iríamos construyendo la línea empezando por la primera palabra y añadiendo las demás en orden, separándolas con espacios de tamaño óptimo b, hasta llegar a una palabra p_a $(a>1)$ que no quepa en la línea, es decir:

$$l_1 + l_2 + ... + l_a + (a-1)*b > L.$$

Si ocurriera que $l_1 + l_2 + ... + l_a + (a-1)*b = L$, esto es, que la palabra encajara perfectamente en la línea, sencillamente imprimiríamos la línea y continuaríamos con la siguiente. Pero si no, necesitaríamos tomar una decisión: o se comprimen las palabras $p_1,...,p_{a-1}$ (recortando el tamaño de los espacios que las separan) para que

pueda caber también p_a en la línea; o bien se pasa la palabra p_a a la siguiente línea y se imprime la línea en curso, aumentando antes los espacios entre las palabras $p_1,...,p_{a-1}$ para que la línea tenga exactamente longitud L.

El algoritmo ávido simplemente va a escoger aquella opción que suponga un menor coste. Obsérvese que estamos ante un típico algoritmo ávido, pues siempre toma su decisión basado en una optimización local y nunca "guarda historia".

Para implementar tal algoritmo, vamos a disponer de un vector que almacena las longitudes de las palabras, y la solución vamos a darla como un vector de registros, uno por cada línea. Cada registro contiene los índices (número de orden) de las palabras que comienzan y terminan la línea, el tamaño de los espacios entre las palabras y el coste de la línea. Esto da lugar al siguiente algoritmo:

```
CONST MAXPALABRAS = ...;
      MAXLINEAS   = MAXPALABRAS; (* para cubrir el peor caso *)
TYPE  REGISTRO=  RECORD
                    primera,ultima:CARDINAL;
                    espacio,coste:REAL;
                 END;
      SOLUCION=  ARRAY [1..MAXLINEAS] OF REGISTRO;
      LONGPALS=  ARRAY [1..MAXPALABRAS] OF CARDINAL;

PROCEDURE Parrafo(L:CARDINAL;n:CARDINAL;b:CARDINAL;VAR l:LONGPALS;
                  VAR sol:SOLUCION):CARDINAL;
    (* L es la longitud de la linea, n el numero de palabras, b el
       tamano optimo de los espacios, l es el vector con las
       longitudes de las n palabras, y en sol almacena la solucion.
       Devuelve el numero de lineas que ha necesitado *)

   VAR  tamanopalabras:CARDINAL; (* long de palabras de la linea *)
        tamanolinea:CARDINAL; (* tamano de la linea en curso *)
        nlinea:CARDINAL;      (* linea en curso *)
        npalabra:CARDINAL;    (* palabra en curso *)
        nespacios:CARDINAL;   (* num. espacios linea en curso *)

   PROCEDURE Espacio(L,tampalabras,nesp:CARDINAL):REAL;
   (* devuelve cero si nesp = 0, o bien un numero mayor que 1 *)
   BEGIN
      IF nesp=0 THEN RETURN 0.0 END;
      RETURN REAL(L-tampalabras)/REAL(nesp);
   END Espacio;

   PROCEDURE ResetContadores(linea,npal:CARDINAL);
   BEGIN
      IF npal<=n THEN (* para la ultima palabra no hacemos nada *)
         sol[linea].primera:=npal;
         sol[linea].coste:=0.0;
```

```
        tamanopalabras:=l[npal];
        tamanolinea:=l[npal];
        nespacios:=0
      END;
    END ResetContadores;

    PROCEDURE Coste(L,b,tampalabras,nesp:CARDINAL):REAL;
      VAR bprima:REAL;
    BEGIN
      bprima:=Espacio(L,tampalabras,nesp);
      IF bprima>REAL(b) THEN RETURN REAL(nesp)*(bprima-REAL(b))
      ELSE RETURN REAL(nesp)*(REAL(b)-bprima)
      END;
    END Coste;

    PROCEDURE CerrarLinea(linea,npal:CARDINAL);
    BEGIN
      sol[linea].ultima:=npal-1;
      sol[linea].espacio:=Espacio(L,tamanopalabras,nespacios);
      sol[linea].coste:=Coste(L,b,tamanopalabras,nespacios);
    END CerrarLinea;
  BEGIN      (* programa principal del procedimiento Parrafo *)
    nlinea:=1;
    ResetContadores(nlinea,1); (* metemos la primera palabra *)
    npalabra:=2;
    WHILE (npalabra<=n) DO
      IF tamanolinea+b+l[npalabra]<=L THEN (* cabe *)
        INC(tamanolinea,b+l[npalabra]);
        INC(tamanopalabras,l[npalabra]);
        INC(nespacios)
      ELSE  (* no cabe de forma optima *)
        IF (tamanopalabras+l[npalabra]+nespacios+1)>L THEN
          (* no cabe en cualquier caso: la pasamos a otra linea *)
          CerrarLinea(nlinea,npalabra);
          INC(nlinea);       (* reinicializamos contadores *)
          ResetContadores(nlinea,npalabra);
        ELSE (* podria caber. Tenemos que tomar una decision *)
          IF Coste(L,b,tamanopalabras,nespacios)>=
          Coste(L,b,tamanopalabras+l[npalabra],nespacios+1) THEN
            INC(npalabra);   (* la metemos en la linea en curso *)
          END;
          (* si no, la pasamos a la otra linea *)
          CerrarLinea(nlinea,npalabra);
          INC(nlinea);
          ResetContadores(nlinea,npalabra);
        END
```

```
      END;
      INC(npalabra);
    END;
    IF sol[nlinea].primera=0 THEN RETURN nlinea-1 END;
    IF sol[nlinea].ultima=0 THEN CerrarLinea(nlinea,npalabra) END;
    RETURN nlinea;
  END Parrafo;
```

La complejidad de este algoritmo es de orden O(n), debido al bucle que se repite a lo más n–1 veces (una por cada palabra menos la primera y aquellas que decidamos meter comprimiendo la línea), y que dentro del bucle todas las operaciones que se realizan son de complejidad constante.

En cuanto a su funcionamiento, desafortunadamente no podemos afirmar que encuentre solución óptima en todos los casos, como pone de manifiesto el siguiente ejemplo.

Supongamos que L = 26, b = 2, y que disponemos de n = 7 palabras, cuyas longitudes son 10, 10, 4, 8, 10, 12 y 12.

El algoritmo anterior, tras meter las dos primeras palabras en la primera línea, tiene que tomar una decisión en cuanto a si la tercera palabra (de longitud 4) debe estar en la primera línea o no. En caso de estar, hay que comprimir los espacios, lo que ocasiona un coste de valor 2; por otro lado, si la pasa a la segunda línea es necesario expandir el espacio entre las dos palabras, lo que supone un coste de valor 4.

Ante esta disyuntiva, el algoritmo decide incluirla en la primera línea, lo que da lugar a la siguiente descomposición en líneas (expresadas con paréntesis):

(10, 10, 4), (8, 10), (12, 12).

El coste global de esta descomposición es 8 (=2+6+0), mientras que si hubiera tomado la alternativa que inicialmente tenía más coste hubiera llegado a la descomposición:

(10, 10), (4, 8, 10), (12, 12)

cuyo coste global es 4 (= 4+0+0).

El motivo del fallo de este algoritmo es su "glotonería", como le ocurre a todos los algoritmos ávidos. En general este problema lo va a tener cualquier algoritmo que, sin disponer de posibilidades de decidir el orden en el que se van produciendo las entradas, no sea capaz de hacer sacrificios locales para obtener resultados globales óptimos.

En cuanto al segundo caso que se plantea en el enunciado de este problema, la situación es mucho más simple ya que no hay que tomar decisiones. O la palabra cabe, o si no hay que pasarla a la siguiente línea pues no se pueden comprimir los espacios entre palabras. El algoritmo que implementa tal estrategia puede obtenerse modificando el anterior:

```
  PROCEDURE Parrafo2(L:CARDINAL;n:CARDINAL;VAR l:LONGPALS;
                     VAR sol:SOLUCION):CARDINAL;
```

```
  (* L es la longitud de la linea, n el numero de palabras, y l es
     el vector con las longitudes de las n palabras. Devuelve el
     numero de lineas que ha necesitado *)

 VAR  tamanopalabras:CARDINAL;
      (* longitud de las palabras de la linea hasta el momento *)
      tamanolinea:CARDINAL; (* tamano de la linea en curso *)
      nlinea:CARDINAL;      (* linea en curso *)
      npalabra:CARDINAL;    (* palabra en curso *)
      nespacios:CARDINAL;   (* num. espacios linea en curso *)

 PROCEDURE Espacio(L,tampalabras,nesp:CARDINAL):REAL;
 BEGIN
   IF nesp=0 THEN RETURN 0.0 END;
   RETURN REAL(L-tampalabras)/REAL(nesp);
 END Espacio;

 PROCEDURE ResetContadores(linea,npal:CARDINAL);
 BEGIN
   IF npal<=n THEN (* para la ultima palabra no hacemos nada *)
     sol[linea].primera:=npal;
     sol[linea].coste:=0.0;
     tamanopalabras:=l[npal];
     tamanolinea:=l[npal];
     nespacios:=0
   END;
 END ResetContadores;

 PROCEDURE Coste(L,tampalabras,nesp:CARDINAL):REAL;
 BEGIN
   RETURN REAL(nesp)*(Espacio(L,tampalabras,nesp)-1.0);
 END Coste;

 PROCEDURE CerrarLinea(linea,npal:CARDINAL);
 BEGIN
   sol[linea].ultima:= npal-1;
   sol[linea].espacio:= Espacio(L,tamanopalabras,nespacios);
   sol[linea].coste:= Coste(L,tamanopalabras,nespacios);
 END CerrarLinea;

BEGIN      (* programa principal del procedimiento Parrafo2 *)
 (* metemos la primera palabra *)
 nlinea:=1;ResetContadores(nlinea,1);
 npalabra:=2;
 WHILE (npalabra<=n) DO
```

```
        IF tamanolinea+1+l[npalabra]<=L THEN (* cabe *)
           INC(tamanolinea,1+l[npalabra]);
           INC(tamanopalabras,l[npalabra]);
           INC(nespacios)
        ELSE  (* no cabe *)
           CerrarLinea(nlinea,npalabra); INC(nlinea);
           ResetContadores(nlinea,npalabra);
        END;
        INC(npalabra);
     END;
     RETURN nlinea;
  END Parrafo2;
```

4.5 LOS ALGORITMOS DE PRIM Y KRUSKAL

Partimos de un grafo conexo, ponderado y no dirigido $g = (V,A)$ de arcos no negativos, y deseamos encontrar el árbol de recubrimiento de g de coste mínimo. Por árbol de recubrimiento de un grafo g entendemos un subgrafo sin ciclos que contenga a todos sus vértices. En caso de haber varios árboles de coste mínimo, nos quedaremos de entre ellos con el que posea menos arcos.

Existen al menos dos algoritmos muy conocidos que resuelven este problema, como son el de Prim y el de Kruskal. En ambos se va construyendo el árbol por etapas, y en cada una se añade un arco. La forma en la que se realiza esa elección es la que distingue a ambos algoritmos.

El algoritmo de Prim comienza por un vértice y escoge en cada etapa el arco de menor peso que verifique que uno de sus vértices se encuentre en el conjunto de vértices ya seleccionados y el otro no. Al incluir un nuevo arco a la solución, se añaden sus dos vértices al conjunto de vértices seleccionados.

En el de Kruskal se ordenan primero los arcos por orden creciente de peso, y en cada etapa se decide qué hacer con cada uno de ellos. Si el arco no forma un ciclo con los ya seleccionados (para poder formar parte de la solución), se incluye en ella; si no, se descarta.

Nuestro objetivo en esta sección no es la de describir en detalle estos algoritmos desde el punto de vista de matemática discreta o la teoría de grafos, sino la de considerarlos desde la perspectiva de los algoritmos ávidos.

a) En primer lugar, nos planteamos la implementación de ambos algoritmos siguiendo el esquema descrito y el análisis de su complejidad (espacio y tiempo).
b) Estos algoritmos trabajan sobre grafos conexos. Nos preguntamos lo que ocurriría si por error se les suministrara un grafo no conexo.

Solución (☞)

a) Para conseguir una implementación sencilla de ambos algoritmos, supondremos que los vértices del grafo ponderado no dirigido $g = (V,A)$ están numerados de 1 a n, así que $V = \{1,2,3,...,n\}$, y que el conjunto de arcos A viene dado por su matriz

de adyacencia ponderada *g*, siendo *g*[*i*,*j*] el peso del arco (*i*,*j*) o bien ∞ si tal arco no existe. Por tanto, vamos a disponer de las siguientes definiciones;

```
CONST n = ...; (* numero de vertices *)
TYPE GRAFO = ARRAY [1..n],[1..n] OF BOOLEAN;
TYPE GRAFO_PONDERADO = ARRAY [1..n],[1..n] OF CARDINAL;
```

Para almacenar el árbol de recubrimiento mínimo (también llamado de expansión), utilizaremos la matriz de adyacencia de un grafo no ponderado.

Comenzaremos implementando el algoritmo de Kruskal, que necesita ordenar los arcos del grafo por orden creciente de peso:

```
PROCEDURE Kruskal(VAR g:GRAFO_PONDERADO; VAR sol:GRAFO);
   VAR  p:PARTICION;
        c1,c2:CARDINAL; (* indican componentes de la particion *)
        g2:GRAFO_ORDENADO;
        i,narcos:CARDINAL; (* numero de arcos del grafo *)
BEGIN
   InicParticion(p);
   narcos:=Ordenar(g,g2);      (* construye g2 a partir de g y *)
   i:=0;                       (* devuelve el numero de sus arcos *)
   WHILE (NOT FinParticion(p)) AND (i<narcos) DO
      (* recorremos todos los arcos *)
      INC(i);
      c1:=ObtenerComponente(p,g2[i].origen);
      c2:=ObtenerComponente(p,g2[i].destino);
      IF c1<>c2 THEN
         Fusionar(p,c1,c2);
         sol[g2[i].origen,g2[i].destino]:=TRUE
      END;
   END
END Kruskal;
```

Veamos los tipos y funciones auxiliares utilizados. En primer lugar, *PARTICION* es un vector que indica a qué componente conexa del grafo pertenece cada vértice, puesto que lo que hacemos es asignar cada vértice a una componente. Cada componente será identificada por el valor de su menor elemento.

```
TYPE PARTICION = ARRAY [1..n] OF CARDINAL;
```

Inicialmente disponemos de todos los vértices del grafo y ningún arco, por lo cual cada uno de los vértices está asignado a una partición distinta (la que constituye el propio vértice aislado). Conforme se van añadiendo los arcos en cada paso del algoritmo el número de particiones va disminuyendo, y los vértices van siendo asignados a las particiones correspondientes. Al incluir un arco que conecta dos particiones, a los elementos de la mayor partición se les asigna el valor de la menor.

Por otro lado, el algoritmo necesita ordenar los arcos del grafo según su peso. Para ello utiliza:

```
TYPE GRAFO_ORDENADO = ARRAY [1..n*(n-1)/2] OF ITEM;
TYPE ITEM = RECORD origen,destino:CARDINAL; peso:CARDINAL END;
```

La función *InicParticion* se necesita para inicializar las correspondientes particiones, constituyendo cada vértice como una partición distinta:

```
PROCEDURE InicParticion (VAR p:PARTICION);
   VAR i:CARDINAL;
BEGIN
   (* cada vertice en una componente distinta *)
   FOR i:=1 TO n DO p[i]:=i END
END InicParticion;
```

Para manejar las particiones, el procedimiento *Fusionar*, como su nombre indica, fusiona dos componentes, asignando a los elementos de la mayor el valor de los elementos de la menor:

```
PROCEDURE Fusionar (VAR p:PARTICION;a,b:CARDINAL);
   VAR i,temp:CARDINAL;
BEGIN
   IF (a>b) THEN (* los intercambiamos *)
      temp:=a; a:=b; b:=temp
   END;
   FOR i:=1 TO n DO
      IF p[i]=b THEN p[i]:=a END
   END;
END Fusionar;
```

La función *FinParticion* comprueba si existe solamente una componente conexa en toda la partición:

```
PROCEDURE FinParticion (VAR p:PARTICION):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO n DO
      IF p[i]<>1 THEN RETURN FALSE END
   END;
   RETURN TRUE;
END FinParticion;
```

Y la función *ObtenerComponente* devuelve el representante de la componente a la que pertenece un vértice:

```
PROCEDURE ObtenerComponente (VAR p:PARTICION; i:CARDINAL):CARDINAL;
```

```
BEGIN
   RETURN p[i]
END ObtenerComponente;
```

Por último, la función *Ordenar* construye un *GRAFO_ORDENADO* a partir del grafo original con todos sus arcos no vacíos ordenados de menor a mayor peso. Esta función devuelve el número de arcos no vacíos que componen el grafo original, y no la incluimos aquí por no extender excesivamente el desarrollo del problema. Para implementarla puede seguirse cualquier método de ordenación:

```
PROCEDURE Ordenar (VAR g:GRAFO_PONDERADO;
                   VAR g2:GRAFO_ORDENADO):CARDINAL;
```

Para el cálculo de su complejidad temporal, veamos el orden de complejidad de las partes que lo componen:

- En primer lugar, *Buscar* es de orden O(1) e *InicParticion* de orden O(n).
- El tiempo de ejecución de las funciones *Fusionar* y *FinParticion* va a depender del número de componentes conexas existentes en la partición pero como en cada paso este número se divide por dos, podemos concluir que su complejidad es de orden O(log$n$).
- Por otro lado, la ordenación de los arcos puede realizarse en un tiempo del orden de O($a$log$a$), siendo a el número de arcos del grafo. Como se verifica que $(n-1) \le a \le n(n-1)/2$ por tratarse de un grafo conexo, su orden es O($a$log$n$).

Resumiendo, el algoritmo consta de una inicialización de orden O($a$log$n$), seguido por un bucle que se repite a veces en donde existen dos operaciones de orden O(log$n$) y varias de orden O(1). Por consiguiente, su complejidad temporal es de orden O($a$log$n$).

Una vez más, es importante hacer notar en este punto que las afirmaciones anteriores se deben a que en esta implementación hemos utilizado el paso de argumentos que sean vectores o matrices por referencia en vez de por valor aun cuando no fuera necesario modificar el valor de tales argumentos. En caso contrario, cada invocación de función supondría una copia de los argumentos a la pila, con el tiempo que eso conlleva.

Respecto a su complejidad espacial, ésta es de orden O(n^2) pues de esta complejidad es la tabla que representa el grafo ordenado. Si en vez de utilizar matrices de adyacencia hubiésemos utilizado una representación no acotada, la complejidad espacial del algoritmo sería de orden O(a) (puesto que sólo hay que almacenar los arcos, y por ser un grafo conexo sabemos que $n-1 \le a \le n(n-1)/2$), aunque quizá se hubiera empeorado la complejidad temporal por el tiempo de acceso asociado a este tipo de estructuras.

Veamos ahora el algoritmo de Prim. Para su implementación vamos a necesitar definir dos tipos especiales:

```
TYPE MASPROXIMO = ARRAY [2..n] OF CARDINAL;
TYPE DISTMINIMA = ARRAY [2..n] OF CARDINAL;
```

siendo *MASPROXIMO*[*i*] el vértice del conjunto de vértices tratados hasta el momento más cercano al vértice *i*, y *DISTMINIMA*[*i*] la distancia desde *i* a ese vértice más próximo. Así, podemos implementar el algoritmo de Prim como sigue:

```
PROCEDURE Prim(VAR g:GRAFO_PONDERADO; VAR sol:GRAFO);
   VAR masproximo:MASPROXIMO;distmin:DISTMINIMA;
       min,i,j,k:CARDINAL;
BEGIN
   InicProx(g,masproximo,distmin);
   FOR i:=2 TO n DO
      min:=MAX(CARDINAL);
      FOR j:=2 TO n DO
         IF (distmin[j]<min) AND (distmin[j]<>0) THEN
            min:=distmin[j]; k:=j
         END
      END;
      sol[k,masproximo[k]]:=TRUE;
      distmin[k]:=0;
      FOR j:=2 TO n DO
         IF (g[j,k]<distmin[k]) THEN
            distmin[k]:=g[j,k];
            masproximo[j]:=k
         END
      END
   END
END Prim;
```

El procedimiento *InicProx* inicializa adecuadamente las variables:

```
PROCEDURE InicProx (VAR g:GRAFO_PONDERADO;VAR v:MASPROXIMO;
                                           VAR d:DISTMINIMA);
   VAR  i:CARDINAL;
BEGIN
   FOR i:=2 TO n DO
      v[i]:=1; d[i]:=g[i,1]
   END
END InicProx;
```

En cuanto a su complejidad, el bucle principal se repite n–1 veces, y los dos más internos también, lo que da lugar a un tiempo de complejidad del orden de $O((n-1)2(n-1)) = O(n^2)$.

Respecto a su complejidad espacial, ésta es también de orden $O(n^2)$ por la representación que hemos utilizado en este caso mediante matrices de adyacencia. En caso de haber utilizado una representación no acotada de los grafos podríamos haber conseguido una complejidad espacial de $O(a)$, aunque quizá se hubiera

empeorado la complejidad temporal del algoritmo por el tiempo de acceso que suponen este tipo de estructuras.

b) Supongamos que suministramos un grafo no conexo como entrada al algoritmo de Kruskal. En primer lugar el algoritmo terminaría puesto que el bucle va recorriendo todos los arcos de tal grafo. Y en segundo lugar su salida sería un árbol de expansión no conexo, pero que si observamos con detenimiento descubriremos que corresponde a la unión de los árboles de expansión mínimos de cada una de las componentes conexas del grafo. En este sentido, el algoritmo es bastante robusto.

No ocurre así con el de Prim, que no funciona en este caso puesto que hace uso de que sea conexo para buscar en cada paso el vértice *k* sobre el cual construir la solución. El que no sea conexo hace que, o bien *k* valga cero y por tanto se indexe erróneamente la matriz solución, o bien no se modifique su valor en cada paso, lo que hace que el algoritmo no termine nunca. Podemos concluir por tanto que el suministrar un grafo conexo como entrada es una precondición fuerte del algoritmo de Prim implementado.

Una vez analizados ambos algoritmos, el uso de uno u otro va a estar condicionado por el tipo de grafo que tratemos. La complejidad del algoritmo de Prim es siempre de orden $O(n^2)$ mientras que el orden de complejidad del algoritmo de Kruskal $O(a\log n)$ no sólo depende del número de vértices, sino también del número de arcos. Así, para grafos densos el número de arcos *a* es cercano a $n(n-1)/2$ por lo que el orden de complejidad del algoritmo de Kruskal es $O(n^2\log n)$, peor que la complejidad $O(n^2)$ de Prim. Sin embargo, para grafos dispersos en los que *a* es próximo a *n*, el algoritmo de Kruskal es de orden $O(n\log n)$, comportándose probablemente de forma más eficiente que el de Prim.

4.6 EL VIAJANTE DE COMERCIO

Se conocen las distancias entre un cierto número de ciudades. Un viajante debe, a partir de una de ellas, visitar cada ciudad exactamente una vez y regresar al punto de partida habiendo recorrido en total la menor distancia posible.

Este problema también puede ser enunciado más formalmente como sigue: dado un grafo *g* conexo y ponderado y dado uno de sus vértices v_0, encontrar el ciclo Hamiltoniano de coste mínimo que comienza y termina en v_0.

Cara a intentar solucionarlo mediante un algoritmo ávido, nos planteamos las siguientes estrategias:

a) Sea (C,v) el camino construido hasta el momento que comienza en v_0 y termina en *v*. Inicialmente *C* es vacío y $v = v_0$. Si C contiene todos los vértices de *g*, el algoritmo incluye el arco (v,v_0) y termina. Si no, incluye el arco (v,w) de longitud mínima entre todos los arcos desde *v* a los vértices *w* que no están en el camino *C*.

b) Otro posible algoritmo ávido escogería en cada iteración el arco más corto aún no considerado que cumpliera las dos condiciones siguientes: (*i*) no formar un ciclo con los arcos ya seleccionados, excepto en la última iteración, que es donde completa el viaje; y (*ii*) no es el tercer arco que incide en un mismo vértice de entre los ya escogidos.

Nos piden implementar ambos algoritmos y probar su funcionamiento, dando ejemplos en donde encuentren solución y en donde fallen, si es que esto ocurre.

Solución (☺)

a) El algoritmo pedido puede ser implementado utilizando los tipos de datos usados en el problema anterior, resultando:

```
TYPE PRESENCIA=ARRAY [1..n] OF BOOLEAN;(* vertices considerados *)

PROCEDURE Viajante1(VAR g:GRAFO_PONDERADO; VAR sol:GRAFO);
(* supone que el recorrido comienza en el vertice 1 *)
   VAR yaesta:PRESENCIA;
       i,verticeencurso,verticeanterior:CARDINAL;
BEGIN
   FOR i:=1 TO n DO yaesta[i]:=FALSE END;
   verticeencurso:=1;
   FOR i:=1 TO n DO
      verticeanterior:=verticeencurso;
      yaesta[verticeanterior]:=TRUE;
      verticeencurso:=Busca(g,verticeencurso,yaesta);
      sol[verticeanterior,verticeencurso]:=TRUE;
   END;
END Viajante1;
```

La clave de este algoritmo está en la función *Busca*, que es la que realiza el proceso de selección, decidiendo en cada paso el siguiente vértice de entre los posibles candidatos:

```
PROCEDURE Busca(VAR g:GRAFO_PONDERADO; vertice:CARDINAL;
                VAR yaesta:PRESENCIA):CARDINAL;
   VAR mejorvertice,i,min:CARDINAL;
BEGIN
   mejorvertice:=1; min:=MAX(CARDINAL);
   FOR i:=1 TO n DO
      IF (i<>vertice)AND(NOT(yaesta[i]))AND(g[vertice,i]<min) THEN
         min:=g[vertice,i]; mejorvertice:=i;
      END
   END;
   RETURN mejorvertice;
END Busca;
```

Respecto a los ejemplos de grafos en donde el algoritmo encuentra o no la solución óptima, comenzaremos por un grafo en donde la encuentra. Sea entonces

	2	3	4

1	1	5	2
2		4	6
3			3

una tabla que representa la matriz de adyacencia de un grafo ponderado g_1 con cuatro vértices. Partiendo del vértice 1, el algoritmo encuentra la solución óptima, que está formada por los arcos

(1,2),(2,3),(3,4),(4,1)

lo que da lugar al ciclo (1,2,3,4,1), cuyo coste es 1 + 4 + 3 + 2 = 10, óptimo pues el resto de soluciones poseen costes superiores o iguales a él: 15, 17, 14, 17 y 10.

Para ver un ejemplo en donde el algoritmo falla, consideraremos un grafo ponderado g_2 con seis vértices definido por la siguiente matriz de adyacencia:

	2	3	4	5	6
1	3	10	11	7	25
2		6	12	8	26
3			9	4	20
4				5	15
5					18

Partiendo del vértice 1 el algoritmo va a ir escogiendo la secuencia de arcos

(1,2),(2,3),(3,5),(5,4),(4,6),(6,1)

lo que da lugar al ciclo (1,2,3,5,4,6,1), cuyo coste es 3 + 6 + 4 + 5 + 15 + 25 = 58. Sin embargo, éste no es el ciclo con menor coste, pues el camino definido por los arcos:

(1,2),(2,3),(3,6),(6,4),(4,5),(5,1)

tiene un coste de 3 + 6 + 20 + 15 + 5 + 7 = 56.

b) El algoritmo pedido en este caso es muy similar al algoritmo de Kruskal que hemos visto en el problema anterior:

```
PROCEDURE Viajante2 (VAR g:GRAFO_PONDERADO; VAR sol:GRAFO);
   VAR  p:PARTICION;
        c1,c2:CARDINAL; (* indican componentes de la particion *)
        g_ordenado:GRAFO_ORDENADO;
        i,narcos:CARDINAL; (* numero de arcos del grafo *)
        u,v:CARDINAL; (* vertices tratados en cada paso *)
        ndest:ARRAY [1..n] OF CARDINAL; (* num. veces que cada
                             vertice es destino en la solucion *)
BEGIN
```

```
    InicParticion(p);
    FOR i:=1 TO n DO ndest[i]:=0 END;
    narcos:=Ordenar(g,g_ordenado);   (* devuelve el num. de arcos *)
    i:=0;
    WHILE (NOT FinParticion(p)) AND (i<narcos) DO
       INC(i);
       u:=g_ordenado[i].origen;
       v:=g_ordenado[i].destino;
       c1:=ObtenerComponente(p,u);
       c2:=ObtenerComponente(p,v);
       IF (c1<>c2) AND (ndest[u]<2) AND (ndest[v]<2) THEN
          Fusionar(p,c1,c2);
          sol[u,v]:=TRUE;
          INC(ndest[u]);
          INC(ndest[v]);
       END;
    END;
 (* ahora solo nos queda el ultimo vertice, que cierra el ciclo *)
    WHILE (i<narcos) DO
       INC(i);
       u:=g_ordenado[i].origen;
       v:=g_ordenado[i].destino;
       IF (ndest[u]<2) AND (ndest[v]<2) THEN (* lo encontramos! *)
          sol[u,v]:=TRUE;
          INC(ndest[u]);
          INC(ndest[v]);
          i:=narcos; (* para salirnos del bucle *)
       END;
    END;
 END Viajante2;
```

Los tipos y funciones utilizados por este procedimiento son los ya vistos en el problema anterior para el algoritmo de Kruskal.

El grafo g_1 es un ejemplo para el cual el algoritmo encuentra la solución óptima, al igual que ocurría con el anterior. Sin embargo, este algoritmo no encuentra la solución óptima en todos los casos, como ocurre por ejemplo con el grafo g_2 del apartado anterior. Para él, y partiendo del vértice 1, el algoritmo va a ir escogiendo la secuencia de arcos

$$(1,2),(3,5),(4,5),(2,3),(4,6),(1,6)$$

que da lugar al mismo ciclo que obteníamos antes, (1,2,3,5,4,6,1), de coste 58 y por tanto no óptimo.

4.7 LA MOCHILA

Dados n elementos $e_1, e_2, ..., e_n$ con pesos $p_1, p_2, ..., p_n$ y beneficios $b_1, b_2, ..., b_n$, y dada una mochila capaz de albergar hasta un máximo de peso M (capacidad de la

mochila), queremos encontrar las proporciones de los n elementos x_1, x_2, ..., x_n ($0 \le x_i \le 1$) que tenemos que introducir en la mochila de forma que la suma de los beneficios de los elementos escogidos sea máxima.

Esto es, hay que encontrar valores (x_1, x_2, ..., x_n) de forma que se maximice la cantidad $\sum_{i=1}^{n} b_i x_i$, sujeta a la restricción $\sum_{i=1}^{n} p_i x_i \le M$.

Solución (☺)

Un algoritmo ávido que resuelve este problema ordena los elementos de forma decreciente respecto a su ratio b_i / p_i y va añadiendo objetos mientras éstos vayan cabiendo.

Para implementar este algoritmo vamos a definir los siguientes tipos y constantes:

```
CONST MAXELEM = ...;  (* numero maximo de elementos *)
TYPE  REGISTRO  = RECORD peso:REAL; beneficio:CARDINAL END;
      ELEMENTOS = ARRAY [1..MAXELEM] OF REGISTRO;
      MOCHILA   = ARRAY [1..MAXELEM] OF REAL;(* composicion final*)
```

Con ellos, el algoritmo ávido para resolver el problema pedido con n elementos y para una capacidad de la mochila M es:

```
PROCEDURE Mochila(VAR e:ELEMENTOS; n:CARDINAL; M:REAL;
                  VAR sol:MOCHILA);
(* supone que los elementos de "e" estan en orden decreciente de
   su ratio bi/pi *)
   VAR peso_en_curso:REAL; i:CARDINAL;
BEGIN
   FOR i:=1 TO MAXELEM DO sol[i]:=0.0 END;
   peso_en_curso:=0.0; i:=1;
   WHILE (peso_en_curso<M) AND (i<=n) DO
      IF (e[i].peso+peso_en_curso)<=M THEN sol[i]:=1.0;
      ELSE sol[i]:=(M-peso_en_curso)/e[i].peso
      END;
      peso_en_curso:=peso_en_curso+(sol[i]*e[i].peso); INC(i)
   END
END Mochila;
```

Respecto al tiempo de ejecución del algoritmo, éste consta de la ordenación previa, de complejidad O($n$log$n$), y de un bucle que como máximo recorre todos los elementos, de complejidad O(n), por lo que el tiempo total resulta ser de orden O($n$log$n$).

Para demostrar que siguiendo la ordenación dada el algoritmo encuentra la solución óptima, vamos a suponer sin pérdida de generalidad que los elementos ya

están ordenados de esta forma, es decir, que $b_i / p_i \geq b_j / p_j$ si $i < j$. Por simplicidad en la notación utilizaremos los símbolos de sumatorios sin los índices.

Sea $X = (x_1,x_2,...,x_n)$ la solución encontrada por el algoritmo. Si $x_i = 1$ para todo i, la solución es óptima. Si no, sea j el menor índice tal que $x_j < 1$. Por la forma en que trabaja el algoritmo, $x_i = 1$ para todo $i < j$, $x_i = 0$ para todo $i > j$, y además $\Sigma x_i p_i = M$. Sea $B(X) = \Sigma x_i b_i$ el beneficio que se obtiene para esa solución.

Consideremos entonces $Y = (y_1,y_2,...,y_n)$ otra solución, y sea $B(Y) = \Sigma y_i b_i$ su beneficio. Por ser solución cumple que $\Sigma y_i p_i \leq M$. Entonces, restando ambas capacidades, podemos afirmar que $\Sigma(x_i p_i - y_i p_i) \geq 0$.

Calculemos entonces la diferencia de beneficios:

$$B(X) - B(Y) = \Sigma(x_i - y_i)b_i = \Sigma(x_i - y_i)p_i\,(b_i/p_i).$$

La segunda igualdad se obtiene multiplicando y dividiendo por p_i. Con esto, para el índice j escogido anteriormente sabemos que ocurre:

- Si $i < j$ entonces $x_i = 1$, y por tanto $(x_i - y_i) \geq 0$. Además, $(b_i/p_i) \geq (b_j/p_j)$ por la ordenación escogida (decreciente).
- Si $i > j$ entonces $x_i = 0$, y por tanto $(x_i - y_i) \leq 0$. Además, $(b_i/p_i) \leq (b_j/p_j)$ por la ordenación escogida (decreciente).
- Por último, si $i = j$ entonces $(b_i/p_i) = (b_j/p_j)$.

En consecuencia, podemos afirmar que $(x_i - y_i)(b_i/p_i) \geq (x_i - y_i)(b_j/p_j)$ para todo i, y por tanto:

$$B(X) - B(Y) = \Sigma(x_i - y_i)p_i\,(b_i/p_i) \geq (b_j/p_j)\Sigma(x_i - y_i)p_i \geq 0,$$

esto es, $B(X) \geq B(Y)$, como queríamos demostrar.

4.8 LA MOCHILA (0,1)

Consideremos una modificación al problema de la Mochila en donde añadimos el requerimiento de que no se pueden escoger fracciones de los elementos, es decir, $x_i = 0$ ó $x_i = 1$, $1 \leq i \leq n$. Como en el problema original, deseamos maximizar la cantidad $\sum_{i=1}^{n} b_i x_i$ sujeta a la restricción $\sum_{i=1}^{n} p_i x_i \leq M$. ¿Seguirá funcionando el algoritmo anterior en este caso?

Solución (☺)

Lamentablemente no funciona, como pone de manifiesto el siguiente ejemplo. Supongamos una mochila de capacidad $M = 6$, y que disponemos de los siguientes elementos (ya ordenados respecto a su ratio *beneficio*/*peso*):

	x_1	x_2	x_3
Peso	5	3	3
Beneficio	11	6	6

El algoritmo sólo introduciría el primer elemento, con un beneficio de 11, aunque sin embargo es posible obtener una mejor elección: podemos introducir los dos últimos elementos en la mochila puesto que no superan su capacidad, con un beneficio total de 12.

4.9 EL FONTANERO DILIGENTE

Un fontanero necesita hacer n reparaciones urgentes, y sabe de antemano el tiempo que le va a llevar cada una de ellas: en la tarea i-ésima tardará t_i minutos. Como en su empresa le pagan dependiendo de la satisfacción del cliente, necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de espera de los clientes.

En otras palabras, si llamamos E_i a lo que espera el cliente i-ésimo hasta ver reparada su avería por completo, necesita minimizar la expresión:

$$E(n) = \sum_{i=1}^{n} E_i \ .$$

Deseamos diseñar un algoritmo ávido que resuelva el problema y probar su validez, bien mediante demostración formal o con un contraejemplo que la refute.

Solución (😐)

En primer lugar hemos de observar que el fontanero siempre tardará el mismo tiempo global $T = t_1 + t_2 + ... + t_n$ en realizar todas las reparaciones, independientemente de la forma en que las ordene. Sin embargo, los tiempos de espera de los clientes sí dependen de esta ordenación.

En efecto, si mantiene la ordenación original de las tareas (1, 2, ..., n), la expresión de los tiempos de espera de los clientes viene dada por:

$$E_1 = t_1$$
$$E_2 = t_1 + t_2$$
.....
$$E_n = t_1 + t_2 + ... + t_n \ .$$

Lo que queremos encontrar es una permutación de las tareas en donde se minimice la expresión de $E(n)$ que, basándonos en las ecuaciones anteriores, viene dada por:

$$E(n) = \sum_{i=1}^{n} E_i = \sum_{i=1}^{n} (n-i+1)t_i \ .$$

Vamos a demostrar que la permutación óptima es aquella en la que los avisos se atienden en orden creciente de sus tiempos de reparación.

Para ello, denominemos $X = (x_1,x_2,...,x_n)$ a una permutación de los elementos $(1,2,...,n)$, y sean $(s_1,s_2,...,s_n)$ sus respectivos tiempos de ejecución, es decir, $(s_1,s_2,...,s_n)$ va a ser una permutación de los tiempos orginales $(t_1,t_2,...,t_n)$. Supongamos que no está ordenada en orden creciente de tiempo de reparación, es decir, que existen dos números $x_i < x_j$ tales que $s_i > s_j$.

Sea $Y = (y_1,y_2,...,y_n)$ la permutación obtenida a partir de X intercambiando x_i con x_j, es decir, $y_k = x_k$ si $k \neq i$ y $k \neq j$, $y_i = x_j$, $y_j = x_i$.

Si probamos que $E(Y) < E(X)$ habremos demostrado lo que buscamos, pues mientras más ordenada (según el criterio dado) esté la permutación, menor tiempo de espera supone. Pero para ello, basta darse cuenta que

$$E(Y) = (n - x_i + 1)s_j + (n - x_j + 1)s_i + \sum_{k=1,k\neq i,k\neq j}^{n} (n-k+1)s_k$$

y que, por tanto:

$$E(X) - E(Y) = (n - x_i + 1)(s_i - s_j) + (n - x_j + 1)(s_j - s_i) = (x_j - x_i)(s_i - s_j) > 0.$$

En consecuencia, el algoritmo pedido consiste en atender a las llamadas en orden inverso a su tiempo de reparación. Con esto conseguirá minimizar el tiempo medio de espera de los clientes, tal y como hemos probado.

4.10 MÁS FONTANEROS

Supongamos que en la empresa del fontanero del apartado anterior aumenta el número de clientes debido a su buena calidad de servicio y deciden contratar a más personal, con lo que disponen de un total de F fontaneros para realizar las n tareas.

Modificar el diseño del algoritmo para que realice la asignación de tareas a fontaneros siguiendo con el criterio de calidad expuesto anteriormente.

Solución (☺)

En este caso también tenemos que minimizar el tiempo medio de espera de los clientes, pero lo que ocurre es que ahora existen F fontaneros dando servicio simultámeamente. Basándonos en el método utilizado anteriormente, la forma óptima de atender los avisos va a ser la siguiente:

- En primer lugar, se ordenan los avisos por orden creciente de tiempo de reparación.

- Un vez hecho esto, se van asignando los avisos por este orden, siempre al fontanero menos ocupado. En caso de haber varios con el mismo grado de ocupación, se escoge el de número menor.

En otras palabras, si los avisos están ordenados de forma que $t_i \leq t_j$ si $i < j$, asignaremos al fontanero k los avisos k, $k+F$, $k+2F$, ...

4.11 LA ASIGNACIÓN DE TAREAS

Supongamos que disponemos de n trabajadores y n tareas. Sea $b_{ij} > 0$ el coste de asignarle el trabajo j al trabajador i. Una asignación de tareas puede ser expresada como una asignación de los valores 0 ó 1 a las variables x_{ij}, donde $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j, y $x_{ij} = 1$ indica que sí. Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador. Dada una asignación válida, definimos el *coste* de dicha asignación como:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} b_{ij} .$$

Diremos que una asignación es óptima si es de mínimo coste. Cara a diseñar un algoritmo ávido para resolver este problema podemos pensar en dos estrategias distintas: asignar cada trabajador la mejor tarea posible, o bien asignar cada tarea al mejor trabajador disponible. Sin embargo, ninguna de las dos estrategias tiene por qué encontrar siempre soluciones óptimas. ¿Es alguna mejor que la otra?

Solución (☺)

Este es un problema que aparece con mucha frecuencia, en donde los costes son o bien tarifas (que los trabajadores cobran por cada tarea) o bien tiempos (que tardan en realizarlas). Para implementar ambos algoritmos vamos a definir la matriz de costes (b_{ij}):

```
TYPE COSTES = ARRAY[1..n],[1..n] OF CARDINAL;
```

que forma parte de los datos de entrada del problema, y la matriz de asignaciones (x_{ij}), que es la que buscamos:

```
TYPE ASIGNACION = ARRAY[1..n],[1..n] OF BOOLEAN;
```

Con esto, el primer algoritmo puede ser implementado como sigue:

```
PROCEDURE AsignacionOptima(VAR b:COSTES; VAR x:ASIGNACION);
   VAR trabajador,tarea:CARDINAL;
```

```
BEGIN
   FOR trabajador:=1 TO n DO (* inicializamos la matriz solucion *)
      FOR tarea:=1 TO n DO
         x[trabajador,tarea]:=FALSE
      END
   END;
   FOR trabajador:=1 TO n DO
      x[trabajador,MejorTarea(b,x,trabajador)]:=TRUE
   END
END AsignacionOptima;
```

La función *MejorTarea* es la que busca la mejor tarea aún no asignada para ese trabajador:

```
PROCEDURE MejorTarea (VAR b:COSTES; VAR x:ASIGNACION;
                       i:CARDINAL):CARDINAL;
   VAR tarea,min,mejortarea:CARDINAL;
BEGIN
   min:=MAX(CARDINAL);
   FOR tarea:=1 TO n DO
      IF (NOT YaEscogida(x,i,tarea))AND(b[i,tarea]<min) THEN
         min:=b[i,tarea];
         mejortarea:=tarea
      END
   END;
   RETURN mejortarea;
END MejorTarea;
```

Por último, la función *YaEscogida* decide si una tarea ya ha sido asignada previamente:

```
PROCEDURE YaEscogida(VAR x:ASIGNACION;
                     trabajador,tarea:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO trabajador-1 DO
      IF x[i,tarea] THEN RETURN TRUE END
   END;
   RETURN FALSE;
END YaEscogida;
```

Lamentablemente, este algoritmo ávido no funciona para todos los casos como pone de manifiesto la siguiente matriz de valores:

Tarea

		1	2	3
	1	16	20	18
Trabajador	2	11	15	17
	3	17	1	20

Para ella, el algoritmo produce una matriz de asignaciones en donde los "unos" están en las posiciones (1,1), (2,2) y (3,3), esto es, asigna la tarea i al trabajador i (i = 1, 2 ,3), con un valor de la asignación de 51 (= 16 + 15 + 20). Sin embargo la asignación óptima se consigue con los "unos" en posiciones (1,3), (2,1) y (3,2), esto es, asigna la tarea 3 al trabajador 1, la 1 al trabajador 2 y la tarea 2 al trabajador 3, con un valor de la asignación de 30 (= 18 + 11 + 1).

Si utilizamos la segunda estrategia nos encontramos en una situación análoga. En primer lugar, su implementación es:

```
PROCEDURE AsignacionOptima2(VAR b:COSTES; VAR x:ASIGNACION);
   VAR trabajador,tarea:CARDINAL;
BEGIN
   FOR trabajador:=1 TO n DO (* inicializamos la matriz solucion *)
      FOR tarea:=1 TO n DO
         x[trabajador,tarea]:=FALSE
      END
   END;
   FOR tarea:=1 TO n DO
      x[MejorTrabajador(b,x,tarea),tarea]:=TRUE
   END;
END AsignacionOptima2;
```

La función *MejorTrabajador* es la que busca el mejor trabajador aún no asignado para esa tarea:

```
PROCEDURE MejorTrabajador (VAR b:COSTES; VAR x:ASIGNACION;
                           i:CARDINAL):CARDINAL;
   VAR trabajador,min,mejortrabajador:CARDINAL;
BEGIN
   min:=MAX(CARDINAL);
   FOR trabajador:=1 TO n DO
     IF(NOT YaEscogido(x,i,trabajador))AND(b[trabajador,i]<min)THEN
        min:=b[trabajador,i];
        mejortrabajador:=trabajador
      END
   END;
   RETURN mejortrabajador;
END MejorTrabajador;
```

Por último, la función *YaEscogido* decide si un trabajador ya ha sido asignado previamente:

```
PROCEDURE YaEscogido(VAR x:ASIGNACION;
                     trabajador,tarea:CARDINAL):BOOLEAN;
   VAR i:CARDINAL;
BEGIN
   FOR i:=1 TO tarea-1 DO
      IF x[trabajador,i] THEN RETURN TRUE END
   END;
   RETURN FALSE;
END YaEscogido;
```

Lamentablemente, este algoritmo ávido tampoco funciona para todos los casos como pone de manifiesto la siguiente matriz de valores:

		Tarea		
		1	2	3
	1	16	11	17
Trabajador	2	20	15	1
	3	18	17	20

Para ella, el algoritmo produce una matriz de asignaciones en donde los "unos" vuelven a estar en las posiciones (1,1), (2,2) y (3,3), con un valor de la asignación de 51 (=16+15+20). Sin embargo la asignación óptima se consigue con los "unos" en posiciones (3,1), (1,2) y (2,3), con un valor de la asignación de 30 (=18+11+1).

Respecto a la pregunta de si una estrategia es mejor que la otra, la respuesta es que no. La razón es que las soluciones son simétricas. Aún más, una es la imagen especular de la otra. Por tanto, si suponemos equiprobables los valores de las matrices, ambos algoritmos van a tener el mismo número de casos favorables y desfavorables.

4.12 LOS FICHEROS Y EL DISQUETE

Supongamos que disponemos de n ficheros $f_1, f_2, ..., f_n$ con tamaños $l_1, l_2, ..., l_n$ y un disquete de capacidad $d < l_1 + l_2 + ... + l_n$.

a) Queremos maximizar el número de ficheros que ha de contener el disquete, y para eso ordenamos los ficheros por orden creciente de su tamaño y vamos metiendo ficheros en el disco hasta que no podamos meter más. Determinar si este algoritmo ávido encuentra solución óptima en todos los casos.

b) Queremos llenar el disquete tanto como podamos, y para eso ordenamos los ficheros por orden decreciente de su tamaño, y vamos metiendo ficheros en el disco hasta que no podamos meter más. Determinar si este algoritmo ávido encuentra solución óptima en todos los casos.

Solución (☺)

a) Supongamos los ficheros $f_1, f_2, ..., f_n$ ordenados respecto a su tamaño, esto es, $l_1 \leq l_2 \leq ... \leq l_n$. Dicho de otra forma, si llamamos L a la función que devuelve la longitud de un fichero dado, lo que tenemos es que $L(f_1) \leq L(f_2) \leq ... \leq L(f_n)$.

El algoritmo ávido indicado en el enunciado de este apartado sugiere ir tomando los ficheros según están ordenados hasta que no quepa ninguno más.

Vamos a demostrar que el número de ficheros que caben de esta forma es el óptimo. Sea m el número de ficheros que dice el algoritmo que caben en un disquete de capacidad d. Si $d \geq \Sigma L(f_i)$ entonces m coincide con n. Pero si $d < \Sigma L(f_i)$, por la forma en la que trabaja el algoritmo sabemos que se verifica la siguiente relación:

$$\sum_{i=1}^{m} L(f_i) \leq d < \sum_{i=1}^{m+1} L(f_i). \qquad [4.5]$$

Sea entonces $g_1, g_2, ..., g_s$ otro subconjunto de ficheros que caben también en el disquete, es decir, tal que

$$\sum_{i=1}^{s} L(g_i) \leq d. \qquad [4.6]$$

Veamos que $s \leq m$. En primer lugar, vamos a suponer sin pérdida de generalidad que el conjunto de los ficheros g_i está también ordenado en orden creciente de tamaño:

$$L(g_1) \leq L(g_2) \leq ... \leq L(g_s).$$

Como ambas descomposiciones son distintas, sea k el primer índice tal que $f_k \neq g_k$. Podemos suponer sin perder generalidad que $k = 1$, puesto que si hasta f_{k-1} los ficheros son iguales podemos eliminarlos y restar la suma de los tamaños de tales ficheros a la capacidad de nuestro disquete.

Por la forma en que funciona el algoritmo, si $f_1 \neq g_1$ entonces $L(f_1) \leq L(g_1)$ pues f_1 era el fichero de menor tamaño. Además, g_1 corresponderá a un fichero f_a en la ordenación inicial, con $a > 1$. Análogamente, g_2 corresponderá a un fichero f_b en la ordenación inicial, con $b > a > 1$, y por tanto $b > 2$, por lo que $L(g_2) \geq L(f_2)$. Repitiendo el razonamiento, los ficheros g_i se corresponderán con ficheros de la ordenación inicial, pero siempre cumpliendo que:

$$L(g_i) \geq L(f_i) \qquad (1 \leq i \leq s). \qquad [4.7]$$

Ahora bien, por la relaciones [4.6] y [4.7] obtenemos

$$d \geq \sum_{i=1}^{s} L(g_i) \geq \sum_{i=1}^{s} L(f_i)$$

Pero entonces, por [4.5], s ha de ser estrictamente menor que $m+1$, y por tanto $s \leq m$, como queríamos demostrar.

b) En este caso el algoritmo no funciona, como pone de manifiesto el siguiente ejemplo: sean (15,10,10,2) los tamaños de cuatro ficheros ($n = 4$) ya ordenados en orden decreciente, y supongamos que disponemos de un disquete con capacidad

$d = 22$. La solución que encontraría el algoritmo ávido es 17 (=15+2), almacenando en el disquete el primer y el último fichero. Sin embargo existe una solución que aprovecha aún más el disquete, la formada por los tres últimos ficheros. Con ellos se ocupa completamente el disquete.

4.13 EL CAMIONERO CON PRISA

Un camionero conduce desde Bilbao a Málaga siguiendo una ruta dada y llevando un camión que le permite, con el tanque de gasolina lleno, recorrer n kilómetros sin parar. El camionero dispone de un mapa de carreteras que le indica las distancias entre las gasolineras que hay en su ruta. Como va con prisa, el camionero desea pararse a repostar el menor número de veces posible.

Deseamos diseñar un algoritmo ávido para determinar en qué gasolineras tiene que parar y demostrar que el algoritmo encuentra siempre la solución óptima.

Solución (☺/☹)

Supondremos que existen G gasolineras en la ruta que sigue el camionero entre Bilbao y Málaga, incluyendo una en la ciudad destino, y que están numeradas del 0 (gasolinera en Bilbao) a G–1 (la situada en Málaga).

Supondremos además que disponemos de un vector con la información que tiene el camionero sobre las distancias entre ellas:

```
TYPE DISTANCIA = ARRAY [1..G-1] OF CARDINAL;
```

de forma que el i-ésimo elemento del vector indica los kilómetros que hay entre las gasolineras i–1 e i. Para que el problema tenga solución hemos de suponer que ningún valor de ese vector es mayor que el número n de kilómetros que el camión puede recorrer sin repostar.

Con todo esto, el algoritmo ávido pedido va a consistir en intentar recorrer el mayor número posible de kilómetros sin repostar, esto es, tratar de ir desde cada gasolinera en donde se pare a repostar a la más lejana posible, así hasta llegar al destino.

Para demostrar la validez de este algoritmo ávido, sean $x_1,x_2,...,x_s$ las gasolineras en donde este algoritmo decide que hay que parar a repostar, y sea $y_1,y_2,...,y_t$ otro posible conjunto solución de gasolineras. Llamaremos X a un camión que sigue la primera solución, e Y a un camión que se guía por la segunda. Sea N el número total de kilómetros a recorrer (distancia entre las dos ciudades), y sea $D[i]$ la distancia recorrida por el camionero hasta la i-ésima gasolinera ($1 \le i \le G–1$). Es decir,

$$D[i] = \sum_{k=1}^{i} d[k] \quad \text{y} \quad D[G–1] = N.$$

Lo que tenemos que demostrar es que $s \le t$, puesto que lo que queríamos minimizar era el número de paradas a realizar. Para probarlo, basta con demostrar que $x_k \ge y_k$ para todo k.

En primer lugar, como ambas descomposiciones son distintas, sea k el primer índice tal que $x_k \neq y_k$. Podemos suponer sin perder generalidad que $k = 1$, puesto que hasta x_{k-1} los viajes son iguales, y en la gasolinera x_{k-1} ambos camiones rellenan su tanque completamente.

Por la forma en que funciona el algoritmo, si $x_1 \neq y_1$ entonces $x_1 > y_1$, pues x_1 era la gasolinera más alejada a donde podía viajar el camionero sin repostar.

Además, también se tiene que $x_2 \geq y_2$, pues x_2 era la gasolinera más alejada a donde podía viajar desde x_1 el camionero sin repostar. Para probar este hecho, supongamos por reducción al absurdo que y_2 fuera estrictamente mayor que x_2. Pero si Y consigue ir desde y_1 a y_2 es que hay menos de n kilómetros entre ellas, es decir,

$$D[y_2] - D[y_1] < n.$$

Por tanto desde x_1 también hay menos de n kilómetros hasta y_2, esto es,

$$D[y_2] - D[x_1] < n$$

puesto que $D[y_1] < D[x_1]$. Entonces el método no hubiera escogido x_2 como siguiente gasolinera a x_1 sino y_2, porque el algoritmo busca siempre la gasolinera más alejada de entre las que alcanza.

Repitiendo el proceso, vamos obteniendo en cada paso que $x_k \geq y_k$ para todo k, hasta llegar a la ciudad destino, lo que demuestra la hipótesis.

El siguiente procedimiento implementa este algoritmo, devolviendo un vector que indica en qué gasolineras ha de pararse y en cuáles no:

```
TYPE SOLUCION = ARRAY [1..G-1] OF BOOLEAN;
PROCEDURE Deprisa(n:CARDINAL; VAR d:DISTANCIA; VAR sol:SOLUCION);
   VAR i,numkilometros:CARDINAL;
BEGIN
   FOR i:=1 TO G-1 DO sol[i]:=FALSE END;
   i:=0;
   numkilometros:=0;
   REPEAT
      REPEAT
         INC(i);
         numkilometros:=numkilometros+d[i];
      UNTIL (numkilometros>n) OR (i=G-1);
      IF numkilometros>n THEN  (* si nos hemos pasado... *)
         DEC(i);               (* volvemos atras una gasolinera *)
         sol[i]:=TRUE;         (* y repostamos en ella. *)
         numkilometros:=0;     (* reset contador *)
      END
   UNTIL (i=G-1);
END Deprisa;
```

4.14 LA MULTIPLICACIÓN ÓPTIMA DE MATRICES

Necesitamos en este problema calcular la matriz producto M de n matrices dadas $M=M_1M_2...M_n$. Por ser asociativa la multiplicación de matrices, existen muchas formas posibles de realizar esa operación, cada una con un coste asociado (en términos del número de multiplicaciones escalares). Si cada M_i es de dimensión d_{i-1}xd_i ($1 \le i \le n$), multiplicar $M_i M_{i+1}$ requiere $d_{i-1}d_i d_{i+1}$ operaciones.

El problema consiste en encontrar el mínimo número de operaciones necesario para calcular el producto M.

En general, el coste asociado a las distintas formas de multiplicar las n matrices puede ser bastante diferente de unas a otras. Por ejemplo, para $n = 4$ y para las matrices M_1, M_2, M_3 y M_4 cuyos órdenes son:

$$M_1(30\times1),\ M_2(1\times40),\ M_3(40\times10),\ M_4(10\times25)$$

hay cinco formas distintas de multiplicarlas, y sus costes asociados (en términos de las multiplicaciones escalares que necesitan) son:

$((M_1M_2)M_3)M_4$	= 30·1·40 + 30·40·10 + 30·10·25	= 20.700
$M_1(M_2(M_3M_4))$	= 40·10·25 + 1·40·25 + 30·1·25	= 11.750
$(M_1M_2)(M_3M_4)$	= 30·1·40 + 40·10·25 + 30·40·25	= 41.200
$M_1((M_2M_3)M_4)$	= 1·40·10 + 1·10·25 + 30·1·25	= 1.400
$(M_1(M_2M_3))M_4$	= 1·40·10 + 30·1·10 + 30·10·25	= 8.200

Como puede observarse, la mejor forma necesita casi treinta veces menos multiplicaciones que la peor, por lo cual es importante elegir una buena asociación. Podríamos pensar en calcular el coste de cada una de las opciones posibles y escoger la mejor de entre ellas antes de multiplicar. Sin embargo, para valores grandes de n esta estrategia es inútil, pues el número de opciones crece exponencialmente con n. De hecho, el número de opciones posible sigue la sucesión de los números de Catalán:

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i) = \frac{1}{n}\binom{2n-2}{n-1} \in \Theta\left(\frac{4^n}{\sqrt{n}}\right).$$

Parece entonces muy útil la búsqueda de un algoritmo ávido que resuelva nuestro problema. La siguiente lista presenta cuatro estrategias diferentes:

a) Multiplicar primero las matrices M_iM_{i+1} cuya dimensión común d_i sea la menor entre todas, y repetir el proceso.

b) Multiplicar primero las matrices M_iM_{i+1} cuya dimensión común d_i sea la mayor entre todas, y repetir el proceso.

c) Realizar primero la multiplicación de las matrices M_iM_{i+1} que requiera menor número de operaciones ($d_{i-1}d_id_{i+1}$), y repetir el proceso.

d) Realizar primero la multiplicación de las matrices M_iM_{i+1} que requiera mayor número de operaciones ($d_{i-1}d_id_{i+1}$), y repetir el proceso.

Queremos determinar si alguna de las estrategias propuestas encuentra siempre solución óptima. Como es habitual en los algoritmos ávidos para comprobar su funcionamiento, sería necesario una demostración formal o bien dar un contraejemplo que justifique la respuesta.

Solución (☺)

Lamentablemente, ninguna de las estrategias presentadas encuentra la solución óptima. Por tanto, para todas es posible dar un contraejemplo en donde el algoritmo falla.

a) La primera estrategia consiste en multiplicar siempre primero las matrices $M_i\,M_{i+1}$ cuya dimensión común d_i es la menor entre todas. Pero observando el ejemplo anterior esta estrategia se muestra errónea, pues corresponde al producto $(M_1M_2)(M_3M_4)$, que resulta ser el peor de todos.

b) Si multiplicamos siempre primero las matrices M_iM_{i+1} cuya dimensión común d_i es la mayor entre todas encontramos la solución óptima para el ejemplo anterior, pero existen otros ejemplos donde falla esta estrategia, como el siguiente:

   Sean $M_1(2\times5)$, $M_2(5\times4)$ y $M_3(4\times1)$. Según esta estrategia, el producto escogido como mejor sería $(M_1M_2)M_3$, con un coste de 48 $(2\cdot5\cdot4 + 2\cdot4\cdot1)$. Sin embargo, el producto $M_1(M_2M_3)$ tiene un coste asociado de 30 $(5\cdot4\cdot1 + 2\cdot5\cdot1)$, menor que el anterior.

c) Si decidimos realizar siempre primero la multiplicación de las matrices $M_i\,M_{i+1}$ que requiera menor número de operaciones, encontraríamos la solución óptima para los dos ejemplos anteriores, pero no para el siguiente:

   Sean $M_1(3\times1)$, $M_2(1\times100)$ y $M_3(100\times5)$. Según esta estrategia, el producto escogido como mejor sería $(M_1M_2)M_3$, de coste 1.800 $(3\cdot1\cdot100 + 3\cdot100\cdot5)$. Sin embargo, el coste del producto $M_1(M_2M_3)$ es de 515 $(1\cdot100\cdot5 + 3\cdot1\cdot5)$, menor que el anterior.

d) Análogamente, realizando siempre primero la multiplicación de las matrices $M_i\,M_{i+1}$ que requiera mayor número de operaciones vamos a encontrar la solución óptima para este último ejemplo, pero no para los dos primeros.